THE UNITED STATES OF AMERICA

TO ALL TO WHOM THESE PRESENTS SHALL COME:

UNITED STATES DEPARTMENT OF COMMERCE

United States Patent and Trademark Office

September 13, 2000

THIS IS TO CERTIFY THAT ANNEXED HERETO IS A TRUE COPY FROM THE RECORDS OF THE UNITED STATES PATENT AND TRADEMARK OFFICE OF THOSE PAPERS OF THE BELOW IDENTIFIED PATENT APPLICATION THAT MET THE REQUIREMENTS TO BE GRANTED A FILING DATE UNDER 35 USC 111.

APPLICATION NUMBER: *60/164,892*
FILING DATE: *November 10, 1999*

PRIORITY DOCUMENT
SUBMITTED OR TRANSMITTED IN
COMPLIANCE WITH
RULE 17.1(a) OR (b)

By Authority of the
COMMISSIONER OF PATENTS AND TRADEMARKS

H. PHILLIPS
Certifying Officer

HARNESS, DICKEY & PIERCE, P.L.C.
ATTORNEYS AND COUNSELORS
P.O BOX 828
BLOOMFIELD HILLS, MICHIGAN 48303
U.S.A.

TELEPHONE
(248) 641-1600

TELEFACSIMILE
(248) 641-0270

Date: November 10, 1999

Hon. Commissioner of Patents and Trademarks
Washington, D.C. 20231

Re: Title: Really Hiding Cryptographic Keys in Software

Atty. Docket: 0722-00004

Sir:

This is a request for filing a provisional patent application. Pursuant to 37 C.F.R. 1.51(c), the following information and documents are provided:

1. The names and addresses of the inventor(s):

First Inventor: Stanley T. Chow
Residence: 3338 Carling Avenue, Nepean, Ontario, K2H 2A8 CANADA

Second Inventor: Harold J. Johnson
Residence: 4 Floral Place, Nepean, Ontario, K2H 6N7 CANADA

Third Inventor: Yuan Gu
Residence: 90 Lightfoot Place, Kanata, Ontario, K2L 3L8 CANADA

Fourth Inventor: _____
Residence: _____

2. A specification having **12** pages.

3. [ ] _____ sheets of drawings showing Figures _____.

4. [ ] This invention was made by an agency of the United States Government or under a contract with an agency of the United States Government under contract number _____.

5. [ ] A Verified Statement Claiming Small Entity Status is enclosed.

6a. [X] A check is enclosed to cover the fees as calculated below. The Commissioner is hereby authorized to charge any additional fees which may be required, or credit any overpayment to Deposit Account No. 08-0750. A duplicate copy of this document is enclosed.

6b. [ ] The fees calculated below will be paid within the time allotted for completion of the filing requirements.

6c. [ ] The fees calculated below are to be charged to Deposit Account No. 08-0750. The Commissioner is hereby authorized to charge any additional fees which may be required, or credit any overpayment to said Deposit Account. A duplicate copy of this document is enclosed.

Atty. Docket No. 0722-000004

Date: November 10, 1999

FILING FEE CALCULATION - BASIC FEE	$150.00
FILING FEE - NON-SMALL ENTITY.....	
FILING FEE - SMALL ENTITY: Reduction by 1/2 ..... A Verified Statement is enclosed.	
Assignment Recordal Fee ($40.00)	
TOTAL	**150.00**

7. [ ] An Assignment of the invention is enclosed. The required cover sheet under 37 C.F.R. §3.11, §3.28 and §3.41 is attached.

8. [ ] Because the enclosed application is in a non-English language, a verified English translation for examination purposes of same [ ] is enclosed [ ] will be filed within the allotted time period.

9. [X] An Express Mailing Certificate is enclosed.

10. [X] Other return postcard

11. Please direct all correspondence and telephone calls relative to this application to the undersigned at the following address:

HARNESS, DICKEY & PIERCE, P.L.C.
P. O. Box 828
Bloomfield Hills, Michigan 48303
(248) 641-1600

If, for some reason, Applicant(s) has/have not paid a sufficient fee, please charge our Deposit Account No. 08-0750 for any further fees which may be due or credit any overpayment to Deposit Account No. 08-0750. A duplicate copy of this document is enclosed.

Respectfully,

Gregory A. Stobbs
Reg. No. 28764

A/PPOV

HARNESS, DICKEY & PIERCE, P.L.C.
ATTORNEYS AND COUNSELORS
P O BOX 828
BLOOMFIELD HILLS, MICHIGAN 48303
U.S.A.

TELEPHONE
(248) 641-1600

TELEFACSIMILE
(248) 641-0270

Date: November 10, 1999

Hon. Commissioner of Patents
and Trademarks
Washington, D.C. 20231

Sir:

EXPRESS MAILING CERTIFICATE

Applicants: Stanley T. Chow, et al

Serial No. (if any):

For: Really Hiding Cryptographic Keys in Software

Docket: 0722-000004

Attorney: Gregory A. Stobbs

"Express Mail" Mailing Label Number.....EJ 179 205 156 US

Date of Deposit.....November 10, 1999

I hereby certify and verify that the accompanying **return postcard, $150 check for filing fee; 2-page transmittal letter (in triplicate); 12-page provisional patent application; along with this Express Mail Certificate** are being deposited with the United States Postal Service "Express Mail Post Office To Addressee" service under 37 C.F.R. 1.10 on the date indicated above and are addressed to the Commissioner of Patents and Trademarks, Washington, D.C. 20231.

Pamela Strauss
Signature of Person Mailing Documents

Really Hiding Cryptographic Keys in Software

Abstract

Many attempts have been made to construct tamper-resistant, secret-hiding software. In particular, many attempts have been made to hide private cryptographic keys in software.They have been notably ineffective. This paper presents a far more powerful approach to the key-hiding, tamper-proofing problem. The material forms part of a number of patent applications.

It is widely believed that tamper-proof, secret-hiding software is impossible. We indicate compelling reasons for changing this belief. We then discuss various approaches to security problems, including the current crop of very weak software-based approaches. We then exemplify a much stronger approach to software-based security, using cryptographic key hiding for the DES cipher as our specific example.

Introduction

> ***"And still it moves."*** — Galileo's comment when told that the planetary motion he had observed with his telescope could not possibly exist, because it would be contrary to established doctrine.

Many attempts have been made to construct tamper-resistant, secret-hiding software, and in particular, to hide private cryptographic keys in software which must perform actual encryption or decryption. They have been notably ineffective. We present here a far more effective approach to the key-hiding, tamper-proofing problem. The methods described form part of a number of patent applications.

There is a variety of uses for hiding cryptographic keys which are used during execution of an application. This is particularly true in connection with a broader strategy for generation of tamper-proof, secret-hiding software. For example, consider the use of biometric information for identification purposes. It is undesirable to transmit biometric information, because it cannot be replaced when it is compromised (each person has a maximum of two thumbs, one voice, two retinas, and so on). If we store biometric information locally, encrypted with a key hidden in software, we can use a strategy over the network which both saves bandwidth (biometric information tends to be bulky) and eliminates risk of compromise due to dissemination of such non-replaceable data.

(The biometric data could not be decrypted by simply extracting the encryption and decryption components from the software, because we would employ subsidiary techniques to make separation into components a very hard problem. Moreover, we would employ subsidiary tamper-proofing, secret-hiding methods to ensure that comparisons of biometric data do not compromise it, even when the attacker has full debugging access, and that the behavior of the application performing such operations is not modifiable in any way useful to the attacker. Hence the biometric information can be well protected both locally and globally.)

There is no sense in even *trying* to do this, however, if it is impossible. Established doctrine, based on an unproven folk-theorem, says that tamper-proof, secret-hiding software *is* impossible: given access to executable code, we can always modify its behavior to suit ourselves or extract its secrets. *The hacker is omnipotent.* After all, a program is just a form of data, which anyone can change at will. Moreover, a program must plainly reveal its semantics, because a program is neither more nor less nor other than an encoding of semantics, readable by a human, a compiler, an interpreter, or a computer of some kind. How, then, can software hide anything significant about itself in a truly effective way?

However widely believed, this doctrine is nevertheless false. There is a widely-known phenomenon which disproves it: but most fail to recognize that it is a counter-example to the folk theorem. Those who have programmed in any large software project know that testing often reveals bugs which we can only fix by

discarding the offending components and *replacing* them with fresh ones, not by *understanding* and *modifying* them. Any attempts to remedy such bugs by modest changes to the offending components result in a seemingly endless variety of new bugs. Such components exhibit a resistance to comprehension or effective change sufficient to frustrate even the most experienced, clever, and patient software experts.

(Of course, in the large software context, the connectivity and opportunities for wide varieties of complex interactions are increased — but massive size is not the *only* way to increase them. A programmer who can truly extract all of the secrets of such a body of software, or bend its behavior to her will using only small incremental changes instead of much larger rewrites, should have a highly lucrative career in upgrading legacy systems, developed at enormous cost, and otherwise destined for the scrap heap. My own decades of experience in large software projects have led me to believe that such folk do not exist.)

Tamper-proof, secret-hiding software is software which exhibits the properties of the stubbornly misbehaving components mentioned above: it exhibits behavior which (1) cannot be understood or reverse-engineered using available time, tools, and human resources, and (2) cannot be modified without producing new unacceptable behavior using available time, tools, and human resources. That is, the software effectively (1) hides its secret data and algorithms, and (2) prevents modifications other than reduction of the program's behavior, or crucial portions thereof, to nonsense.

The problem of generating tamper-proof, secret-hiding versions of arbitrary programs is indeed a massive one, far beyond the scope of a single paper. Approaches required depend on the nature of the data (integral or floating; scalar, arrayed, or otherwise structured; internal to the program or external in files and other media), and the nature of the control strategy (object-oriented, procedural, or functional; serial, implicitly parallel, or explicitly parallel in various ways). Here, we focus specifically on the problem of hiding a DES (Data Encryption Standard) key in software, that is, of producing fully standard DES encryption and decryption functions which use a secret key which is exceedingly difficult to extract from the functions. The (patent pending) techniques used have high overheads compared to previous approaches. However, they are also enormously more effective in hiding a secret key and frustrating targeted tampering.

One might think that tamper-proofing, in the sense of creating software which changes behavior drastically in response to small changes, is irrelevant to secret-hiding, such as hiding a cryptographic key. Actually, it is quite relevant: it makes perturbation-based analysis (analysis by examination of responses to small changes) much more difficult.

Previous Approaches Are Either Expensive or Weak

Many approaches have been taken to the problems of secret-hiding and tamper-proofing:

Hardware approaches have been proposed in profusion. These are generally inapplicable to the installed base of computers, and would be (or are) costly to administrate and deploy.

A highly elaborate hardware-based approach — obviously totally inapplicable to the installed base of computers — which hides instructions, data, and control flow, is presented in [OST 1992]. A hardware approach conceptually similar to a safe with a combination lock is presented in [MARX 1998].

Among hardware-based approaches, we have *dongles* and *smart cards*, which move data and code inside a physical device. They are costly for administration and transport, compared to software-based approaches, where manufacture is virtually free and transport can be electronic. Moreover, due to structural limitations, *smart cards* have been far more vulnerable to penetration of their secrets than was hoped: news items describing incidents of penetration have been appearing on a regular basis.

There have been many special-purpose tricks to prevent unauthorized software copying of software, including start-up code examining supposedly unused parts of an attached hard-disk, 'fingerprinting' particular PC environments, and querying hardware, operating-system-provided, or network-provided identi-

fiers. They have been defeated by hackers on a regular basis. As is well known, special-purpose copy programs which casually remove copy-protection from PC software in transit are available for a fairly modest price if you know where to ask.

A software approach for computing with encrypted data is described in [AHI 1987]. This method hides the actual value of the data from the software doing the computation. The computations which are practical using this technique are restricted, of course. (For example, it is not well suited to DES key hiding.) In terms of obfuscation, it is far superior to anything generally available in a commercial obfuscator.

[COL 1998-1] provides a method for concealing the intent of the control flow, and [COL 1998-2] provides more comprehensive proposals on obfuscation, together with methods for obfuscation of structured and object-oriented data. Again, these approaches are far superior to what is generally available in commercial obfuscation tools.

There remains a weakness, however, even in the methods proposed by [AHI 1987], [COL 1998-1], and [COL 1998-2]. *Obfuscation* and *tamper-proofing* are distinct problems. For example, consider removing password protection from an application by changing one branch from a conditional one to an unconditional one. Plainly, this vulnerability cannot be eliminated effectively by any amount of mere obfuscation — a patient hacker tracing the code will eventually find the "pass, friend" / "begone, foe" branch. We need other methods to avoid *single points of failure* for tamper-proofing. The above methods are quite good for *obfuscation*, but they are not nearly as effective for *tamper-proofing*.

Existing general-purpose commercial software obfuscators use a variety of techniques including: removal of debug information, changing variable names, merging or splitting live ranges of variables, introducing irreducible flow graphs, and (particularly in the case of Java) modifying code structures to avoid stereotyped forms for source control structures. They have minimal effect on data- and control-flow graphs as revealed in internal forms used for data-flow analysis by optimizing compilers or program slicing tools. Irreducibilities can be handled by well-known compiler techniques. Information present at the 'machine' code level, or equivalent, is not obscured at all, including the data used in computation. For example, information about DES encryption and decryption, and probably any reasonably secure form of encryption or decryption, cannot be hidden effectively using techniques such as these.

An alternative approach is to encrypt the program either as a whole or in parts, and then to decrypt the program or its components temporarily as they are needed. (See [AUC 1996], for example.) This exposes executable images of the program or its components to logic analyzers and the like, permitting recovery of the original program either in entirety, or in a piecemeal fashion as its components are exercised. Moreover, unless this paper's subject problem of hiding cryptographic keys is also solved, the key can be extracted from the software and used to decrypt the entire program.

The above techniques have the advantage that the size of the code, including any encrypted code, is either not increased or increased but little. However, the obfuscation obtained is plainly quite weak, since the executed code, control- and data-flow-analyzed in graph form, is either isomorphic to, or nearly isomorphic to, the unprotected code.

We could also hide the real code by introducing dummy code — say, by making every other statement a dummy statement, designed to look much like the real thing. Despite its higher overhead, this approach has two fatal weaknesses: (1) It is vulnerable to data-flow analysis (DFA) to discover the redundant dummy code, unless extra care is taken to introduce large amounts of aliasing in the dummy code to foil DFA. (2) Even if DFA can be rendered ineffective, if x% of the code is dummy code, then $100 - x$% of the code is significant. For realistic values of x, a patient attacker can locate which statements matter and which don't by trial and error.

Attempts have been made to hide cryptographic keys by a variety of more specific approaches, including

splitting the key into pieces (but the pieces can be reassembled by tracing execution), and modifying the algorithm to use a disguised key (but human capacities limit the amount of disguise to a small number of algorithmic steps) or a disguised algorithm (but again, with a similar weakness due to the restrictions imposed by our limited human capacities).

In addition, a variety of cryptographically weak approaches have been used for encryption and decryption, to avoid the use of any explicit key whatever. These are vulnerable either to a cryptographic black-box attack (if a plain-text can be recognized in an automated way) or to algorithmic analysis with the aid of debugging tools (since the would-be encryption is then a data transformation of quite limited algorithmic complexity).

In general, then, the state of the art has been that programs could not be made effectively secret-hiding and tamper-proof. Specifically, cryptographic keys for reasonably secure ciphers could not really be hidden in software.

And Now For Something Completely Different!

In addition to the widely-recognized principle of *obscurity* in software protection, our over-all approach to tamper-proof, secret-hiding software involves the following principles:

1. *Targeting:* We suit the approach specifically to the operations to be performed and the data to be manipulated.
2. *Partial evaluation:* Part of the process of hiding constant input data is to partially evaluate the application with respect to that data. (In the case of DES key-hiding, for example, the key is constant and is eliminated by partial evaluation.) This principle is allied to the principle of *diffusion*.
3. *Fusion:* Encoded software handles the data in such a way that multiple components are manipulated together, so that separating out individual original (i.e., pre-encoding) data operations is difficult, and tampering with one entity in effect modifies the behavior of more than one entity.
4. *Diffusion:* Encoded data and computation distribute information among multiple sites, so that no site alone is sufficient for understanding, ambiguity is increased, and tampering at individual sites is made less effective.
5. *Fake robustness:* Presumably, *true* robustness would preserve the *same* computation even after some forms of tampering. We 'fake' such robustness by avoiding failure responses to data in the presence of tampering. Instead, computation proceeds with apparent normalcy, but along nonsensical lines. This is strongly allied to the principle of *anti-holographic behavior*.
6. *Anti-holographic behavior:* Tampering with a small part of a hologram causes a slight reduction in resolution. We seek the opposite behavior, where the effect of any small change is to produce *large, widespread, cascading* changes in behavior.

We now proceed to describe our approach for DES encryption and decryption, in accordance with the above principles.

Targeting for DES

DES inputs a 64-bit block to be encrypted or decrypted and a 64-bit raw key (of which only 56 bits are actually used: the low-order bit of each raw key byte is discarded, or can be used for parity), and outputs a 64-bit result.

A description of (single) DES is provided in [FIPS 46-3]. A description and an extensive discussion are provided in [SCH 1996].

There are only three kinds of data operations in DES:

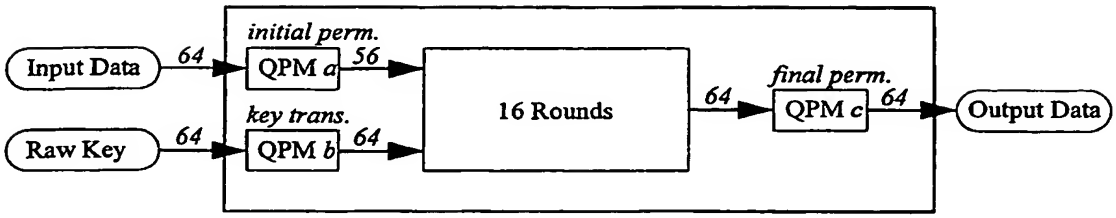

Figure 1: Outer Structure of DES

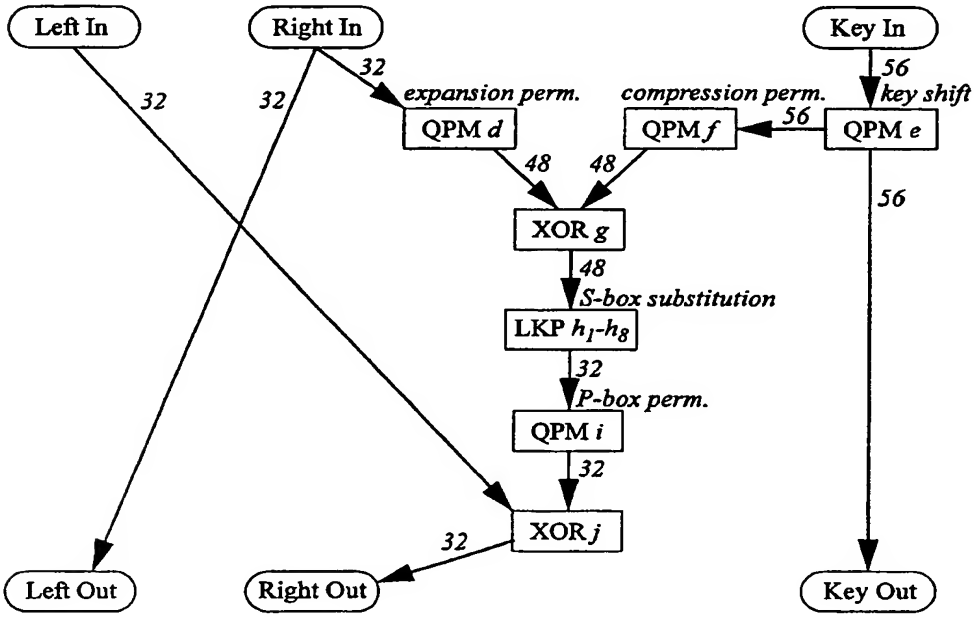

Figure 2: Structure of One DES Round

1. Selecting some or all bits from a bit-string and re-ordering them into a new bit-string, possibly with multiple appearances of certain bits. [SCH 1996] refers to these as *permutations*, noting that this is not quite accurate since they are not necessarily bijections. Let us refer to such transformations, in general, as *quasi-permutations* (QPMs), with the true *permutations* being the special case where the QPM is a bijection. Each QPM operation is controlled by a table which for each *to*-bit of the output bit-string gives the *from*-bit in the input bit-string whose value it has, except for key-shift QPMs, which are simple rotation permutations, each of which is described by a simple signed shift count.
2. Bit-wise exclusive or (XOR).
3. Looking up elements in a table (LKP). In DES, before we perform any transformations, these are look-ups in 64-element tables of 4-bit-strings (each of which is called an *S-box* — S for "substitution"), using a 6-bit-string as an index. Initially, each LKP operation is controlled by one of eight *S-box* tables indicating the substitutions it is to perform.

Figure 1 shows the outer structure of DES. We use a specialized presentation designed to emphasize the three basic kinds of operations making up DES. Italicized numbers adjacent to the arrows indicate the bit-widths of the indicated values. QPM *a* and QPM *c* are true *permutations* (no omissions, no duplicated bits), with QPM *c* being the inverse of QPM *a*. (QPM *a* is called the *initial permutation*, and QPM *c* is called the *final permutation*.) QPM *b* (the *key transformation*) selects 56 of 64 bits from the raw key, and rearranges the bits. The outer box represents the entire DES function (whether encryption or decryption). The inner structure of DES comprises 16 rounds of processing, which are identical except for one minor variation in the final round and the variations in one of the internal QPM operations (namely, QPM *e*, the *key shift*).

Figure 2 shows the internal structure of one of the 16 DES rounds. Left In and Right In are the left and right halves of the data being processed as it enters the round, and Left Out and Right Out are these halves after the processing performed by the rounds. Key In is the 56-bit key as it enters the round, and Key Out is the 56-bit key as it leaves the round. QPM *d* (the *expansion permutation*) repeats certain bits, whereas QPM *f* (the *compression permutation*), which produces the round sub-key as its output, omits certain bits. QPM *e* (the *key shift*) consists of rotations of the left and right halves of the 56-bit key by an identical amount, in a direction and with a number of shift positions determined by the round number and by whether encryption or decryption is being performed. LKP h_1-h_8 (performing *S-box substitution*) are the eight S-box lookups performed in the round. (In the DES standard, the indices for the LKP operations h_1-h_8 are each, in effect, preceded by yet another QPM operation, which permutes the six input bits so that the low-order or right-most bit becomes the bit second from the left in the effective index, but this QPM can be eliminated to match what we have shown above by re-ordering the elements of the S-box tables.) QPM *i* (the *P-box permutation*) permutes the results of LKP h_1-h_8, presumably to accelerate diffusion of information across all bits.

All rounds are performed identically except for the previously mentioned differences in QPM *e* (the *key shift*) and the swapping of Left Out and Right Out (relative to what is shown in Figure 2) in the final round.

Due to the prevalence of QPM operations, we handle DES operations at the level of individual Boolean values. At that level, the original QPM operations no longer appear as operations in the data-flow: instead, they simply determine connectivity of the LKP and XOR operations. Note that none of the operations can terminate abnormally, irrespective of their inputs, but changing the inputs or the operation changes the result, so our implementation is completely *fake robust*. Moreover, as with many ciphers, slight changes produce cascading, wide-spread behavioral changes, so that it exhibits *anti-holographic behavior* (which we will magnify).

We also note that the 48 bits emitted by QPM *f* (the *compression permutation*) are entirely determined by the original key and the round number, since no information travels from the data-portion of the round to the key-portion of DES. Hence we unroll the rounds-loop completely, leaving only a directed acyclic graph of Boolean operations. In order to avoid multiple-output operations, and to facilitate optimization, we replace the eight S-box lookups, LKP h_1-h_8, with 32 T-box lookups, LKP k_1-k_{32}. T stands for "tiny", since only one bit is emitted per T-box. If we regard the bits of the S-box elements as columns in a Boolean or bit matrix, then each T-box is one column of the corresponding S-box. LKP k_1-k_4 represent LKP h_1, with each output representing one bit of the original h_1 output; LKP k_5-k_8 represent LKP h_2, and so on. We make the T-box lookups in different rounds independent of one another, by copying their tables, so that one round can be modified without affecting the others. We then have 16 rounds worth of 32 T-box tables each: 512 independent T-box tables in all, each containing 64 Boolean elements (since each has six Boolean inputs).

After the above-described targeting of the operations in DES, the initial connections surrounding one T-box operation, LKP k_i, appear as shown in Figure 3.

One further note on targeting: we note that DES is vulnerable to attack from the ends (the beginning and end of encryption or decryption; see [SCH 1996]). Hence we note at this point that we need some way to

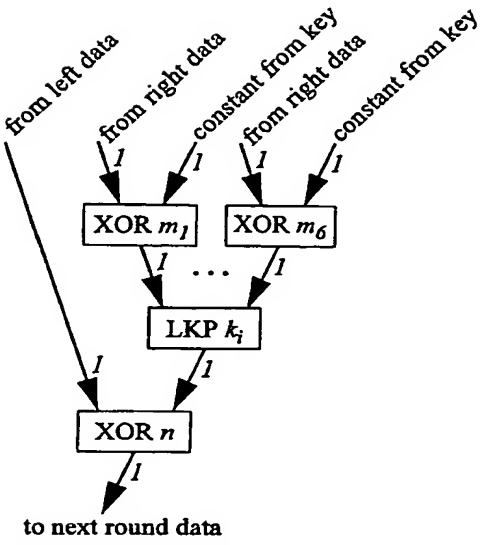

Figure 3: Initial Connections of One T-box Operation

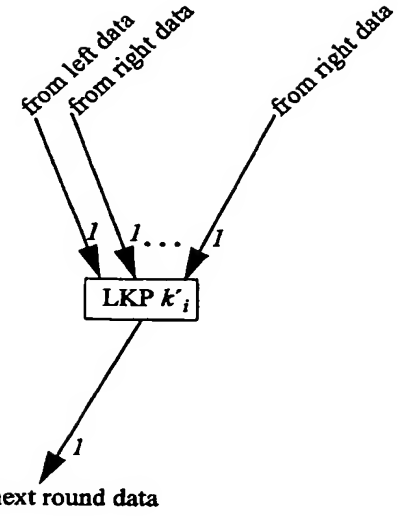

Figure 4: T-box Connections After Partial Evaluation

increase the protection of the initial and final portions of our implementation.

Partial Evaluation with Respect to the Key

We note in Figure 3 that the right operands of XOR m_1-m_6 are constants. Hence we can delete the right operands and replace each XOR with a Boolean identity (if the constant is **false**) or a Boolean NOT operation (if it is **true**). We can then eliminate these identities and NOT operations by connecting the LKP inputs directly to the operations previously providing inputs to the identities and NOTs. To do this, we must adjust the contents of the LKP table to allow for the effect of any inputs resulting from the elision of a NOT operation. (This is easily accomplished by re-ordering the elements of the tables.)

We then further reduce the operation count by folding the XOR shown as XOR n in Figure 3 above together with the LKP shown as LKP k_i above. To do this, we add a new leftmost input to LKP k_i and connect it to the input to XOR n. We then eliminate XOR n. We take the elements of LKP k_i's table, make a copy but with every element inverted, and concatenate that to the end of the original table. The result is that the new LKP (LKP k'_i, say) now includes the effect of XOR n (thereby increasing the degree of *fusion* in our implementation) and we now have a version of DES consisting of 512 7-input T-box lookup operations, connected together. The connections of a typical LKP operation, after partial evaluation, are shown in Figure 4.

Information from the key and the manipulations of the key has now been *diffused* into the T-box LKP operations, so we have started to satisfy our principle of *diffusion* of computations and data.

The key has now been eliminated from the computation. Note, however, that the connections of a T-box LKP from the second previous round, created by eliding the XOR operations of XOR n (see Figure 3) give away the identities of the T-box LKPs. We will deal with this problem in a later transformation.

Padding the DES Implementation

As part of our targeting, we noted the vulnerability of DES to attacks starting at the beginning and the end of the computation. To address this vulnerability, we *pad* the DES implementation.

We generate a series of DES-based identities. A DES-based identity is created using DES implemented exactly as described above, except that we omit the effects of the *initial* and *final permutations* shown as QPM *a* and QPM *c* in Figure 1, and we may cut the number of rounds from 16 to some smaller number (as low as two). Composition of a pair of these, performing complementary encryption and then matching decryption with the same randomly-chosen key, produces a DES-based identity. Each such identity takes in 64 bits and emits the same 64 bits with no change.

We pad our implementation by injecting such identities (based on various randomly chosen keys) at various points within our series of unrolled rounds, focussing on the beginning and end of our implementation. Since they are identities, they have no effect whatever on the results of our implementation. At this point, they do not sound like a sensible addition (after all, identical lineups of 64 Booleans are likely to stick out during tracing of the software). However, after further techniques have been applied, they are identities no longer.

These identities, even after further processing, continue to have the property inherent in DES that interfering with any individual Boolean, or any computation which produces such a Boolean, has a diffuse effect, altering many bits in future rounds. Hence this padding is *not* dummy code. It changes what happens in response to tampering: it contributes to *anti-holographic behavior.* That is, any small change will have an increased tendency to produce a wide-spread, cascading effect over many output bits; even more so than in ordinary DES (with respect to the 'real' rounds). We also address the specific need to protect the beginning and end of the computation. Pads also increase the *obscurity* of the implementation: there is no longer just one key for an attacker to identify in any given hidden-key cryptographic function: there are several.

We may inject further pads at points in the middle of the computation to increase anti-holographic behavior as much as we wish. At a minimum, we should enclose a sequence of initial round pairs (one or more) between two pads, and similarly for a sequence of final round pairs.

Diffusing Information Among T-boxes

Here we perform further transformations in accordance with our principle of *diffusion* of computations and data.

For this transformation, we repeatedly take one T-box (*original*) and create two corresponding T-boxes (*left* and *right*), with the same inputs as *original*. (The *original* T-box must not be a final output T-box.) The tables for *left* and *right* are computed so that *left* does not match *right*, and neither matches *original*. We first present a simple, somewhat weak approach, and then recommend a refinement which makes the approach much stronger. Here is the simplified method:

We choose a Boolean function with two inputs and one output. There are 16 of these, but we do not use Boolean functions for which some input is a '*don't care*'. There are six functions we must therefore reject: namely, those which output constant **true**, constant **false**, the left input, the right input, **not** the left input, and **not** the right input. The remaining ten Boolean functions are usable, and we choose among such functions at random for any given *left, right* pair. Let us denote the function we choose for any particular *left, right* pair by "*func*".

We fill the tables for *left* and *right* as follows:

For each element indexed by *i* in *original*, where *i* ranges from 0 to 127 inclusive (since the T-boxes have seven inputs at the start of this transformation), we choose a pair of values *x,y* for the *left* and *right* elements indexed by *i*, respectively, such that *func*(*x,y*) has the same value as element *i* of *original*. There are

often multiple choices of Boolean *x,y* value pairs which achieve this, and we choose randomly among such choices. Hence information from *original* is randomly *diffused* between *left* and *right*, with the addition of random, redundant information.

For any T-box LKP operations which input the value of *original*, we instead make them input values from both *left* and *right*. This makes them *n*-input T-box LKPs, where $n \geq 8$. (Initially, $n = 8$, but as we proceed, *n* may take on higher values with the splitting of more inputs into input pairs.)

The Boolean value stored in the table of a resulting 8-input T-box LKP operation for any 8-bit input vector is determined as follows: Let *A* and *B* be bit-strings with a combined length of six bits, and let *u*, *v*, and *w* be individual bits. We represent concatenation by juxtaposition. For any element indexed by some index *AuvB* in the expanded 8-input table, the value stored is the same as that of the element indexed by *AwB* in the 7-input table from which it is derived, where $w = func(u,v)$. We handle $n > 8$ similarly.

By working this transformation backwards from the output T-box LKP operations to the beginning of the DES implementation graph, we can arrange that, in general, T-box LKP operations other than those producing the final outputs and the initial ones whose inputs are not from other T-box LKP operations, have more than seven inputs.

This transformation is quite simple, and contributes greatly to *obscurity*, by diffusing information among T-box LKP operations and thereby making their contents randomly perturbed relative to their original contents. Moreover, it tends to make the injected *pad* identities not quite identities anymore.

However, we can make it combinatorially stronger (that is, increase the number of possible functions above ten), and at the same time tackle the T-box identification problem mentioned at the end of the section on partial evaluation, with the following refinement:

Instead of having *func* be a function of only two Boolean inputs, we make it a function of three Boolean inputs, where one of the inputs is one of the inputs of *original* which comes from the round previous to *original*. Let us call this extra input *p*. Then *func* must be a Boolean function such that (1) if there is a '*don't care*' input, it is the *p* input, and (2) for each value of *p*, it is possible to make *func* return either **true** or **false** by modifying the other inputs. This increases the number of choices for *func* from ten to 100. Then LKPs which used to input from *original* input from all of: *left*, *right*, and the LKP, or the original input from the start of DES, which is the source of *p*.

We then replace our uses of $func(x,y)$ and $func(u,v)$ above with uses of $func(p,x,y)$ and $func(p,u,v)$, respectively. Filling in the table for the expanded input set is a straightforward extension of the methods used above. In addition to increasing the combinatorial complexity of determining the contents of the diffused tables in *left* and *right*, this refinement makes it much harder to identify which T-box LKP corresponds to which column of which S-box, since connections from two rounds back become more frequent, and this plus a later T-box LKP input permutation step make T-box LKP identities ambiguous.

It is important to make suitable choices for *original* and for the source of *p*. Examination of the interconnection pattern for T-box LKP operations will show that in many cases we can make the identity of a T-box LKP with respect to a column in an S-box *ambiguous*, by increasing the number of inputs from two rounds previous from one to two or more, so that it is not clear which input from the second previous round came from eliding an XOR (XOR *n* in Figure 3) and which was added by the *diffusion* transformation. When this is combined with the transformation described in the next section of this paper, which permutes the T-box LKP inputs, it makes identification of T-box LKPs with their corresponding S-box columns far more difficult, combinatorially speaking. The details depend on the nature of the *expansion permutation* (QPM *d* in Figure 2) and the *P-box permutation* (QPM *i* in Figure 2), which together determine the connectivity among rounds.

The above approach, with or without the recommended refinement, easily extends from producing *left*,

right pairs of T-box LKP operations to producing triplets — *left, middle, right* — or even quadruplets or larger numbers. We can also increase the number of inputs in non-initial T-box LKP operations, either by producing more pairs, or by producing triplets or quadruplets instead of pairs, or by some combination of these approaches. We can also vary the number of inputs among T-box LKP operations, making the structure of our DES implementation highly irregular.

Encoding T-box Input Vectors

In our next step, we encode the input vectors to T-box LKP operations. At this point, the T-box operations have 7- or 8-bit input vectors (or, optionally, larger ones). The encoding consists of (1) flipping randomly chosen bits and (2) permuting the positions of the vector elements.

First, we perform the flipping part of the encoding. We randomly select inputs for inversion. We do this only where the sources of these inputs are internal to the implementation; that is, we do not flip any bits in the input data. When a bit is flipped, the bits of its source T-box's table are inverted.

Inputs to T-boxes may come from shared sources. As a result, when two T-boxes disagree on the encoding of inputs coming from the same other T-box, that source T-box is no longer fully sharable (since its output must be delivered to one client flipped and to another unflipped). As a result, this stage increases the number of T-box LKP operations in the implementation.

The second part of the coding is to randomly permute the inputs of each T-box LKP operation. We re-order the elements of each T-box LKP table to allow for the new arrangement of the inputs.

These modifications to the T-box LKP tables intermingle elements which previously were widely separated, increasing the degree of *fusion* in our implementation. They also increase the *obscurity,* as does the presence of multiple T-boxes derived from one T-box, and containing different tables. Moreover, the previously described *pad* rounds injected into our implementation have now very definitely ceased to be identities.

Generating an Executable DES Encryption or Decryption Routine

Up to this point, we have dealt with a symbolic Boolean DAG, which is not in a form suitable for execution on any platform. Note that the DAG consists entirely of T-box LKP operations. A straightforward implementation of DES based on the DAG, then, is as follows:

Each LKP operation is represented by a call to a utility function. For an n-input LKP, it takes $n + 1$ arguments. The extra argument is a pointer to the table of Booleans to be used for that particular LKP operation. The utility function compresses its inputs into an index, indexes into its table to find the result, and returns that result.

The body of the DES function, then, consists of an initial expansion of the 64-bit input data block into 64 separate values, followed by a chain of T-box LKP routine calls, plus any needed loads and stores, implementing the desired Boolean DAG's connectivity, followed by a compression of the 64 result Booleans into a 64-bit result value, which is returned.

We can reduce the number of arguments to each of the above LKP routine calls by one, by taking advantage of the fact that the calls are chained together in a specific sequential order. Therefore, we can sequence through the tables used in the successive calls by having the utility routines index through a sequence of tables stored in just that sequential order. Thus, the tables can be implicit in the calls, instead of being passed as an argument in each call. We would then begin the body of the DES function by setting the appropriate starting state for iterating through these tables.

In Practice, Plain-Text Is Encoded

Although we described an implementation in which the DES implementation is standard, in practice, we would use an encoded implementation in which encrypted information is decrypted to an encoded format with bits permuted and some bits flipped. In many contexts, we can compute with such encoded data, and using an encoded plain-text form makes the problem of penetrating the DES implementation to find the key significantly harder.

How Hard Is It to Find the Key?

We have introduced a new way to generate an implementation of DES with an implicit, hidden key. It is intended for use where key-hiding is important, but the volume of data to be encrypted or decrypted is modest, so that we can tolerate a much slower implementation in order to achieve a greatly increased level of security. Our approach injects a huge amount of random, arbitrary information into the structure of the hidden-key DES implementation.

At present, there are, quite simply, *no* widely-accepted, theoretically well founded metrics for estimating the level of security delivered by a technology for the production of secret-hiding, tamper-proofing software. This is a vast unexplored area in the theory of computational complexity.

Any theory covering such an area must deal with the computational complexity of moving from one level of abstraction to another. It must provide a basis for comparing a distance between two programs in terms of their respective *codes*, to a distance between them in terms of their respective *behaviors*, so that the effects of tampering can be gauged. (For a tamper-proof, secret-hiding program, we would expect tampering which traverses only a small distance by the *code* metric to produce a behavioral change which traverses a huge distance by the *behavior* metric; that is, we would expect *anti-holographic behavior.*)

We most eagerly await theoretical and practical work in this area!

The way S-box LKP operations are interconnected when we unroll the round loop of DES determines the way the T-box LKP operations are interconnected when we convert from S-box LKPs to T-box LKP operations and then partially evaluate with respect to the key. (The key has no effect on connectivity.) This would allow us to identify the T-box LKPs. However, after we perform the refined version of diffusion of information into pairs of T-box LKP tables, followed by encoding of T-box LKP input vectors, which permutes the inputs, T-box LKP identification is ambiguous. While analysis of the combinatorial complexity of T-box identification is a dauntingly difficult undertaking, it seems clear that these transformations make the problem of identifying the effective positions of individual T-box LKP operations, as compared to columns of the original S-boxes, a combinatorially sizable search problem.

Even if we could identify all of the T-box LKPs with individual columns in individual original S-boxes, however, we would still not know the key. Due to padding, an attacker must contend with multiple keys and unknown boundaries between pad rounds and 'real' rounds. The difficulty of finding the 'real' key can be increased by using more injected pads, or pads with more rounds, or both. Due to the encoding and diffusing of information among tables, and due the large size of the combinatorial search for ways to disambiguate T-box LKP identities and then recombine diffused pairs of LKPs back into single LKPs, we believe that we have created an extremely large combinatorial guessing problem to find the key. We can make this problem harder by increasing the number of inputs in the T-boxes — thereby making more and more T-box tables the result of combinatorially hidden diffused information. Exact computation of the search problem's complexity is difficult, even if most T-box LKPs have only eight inputs, but the combinatorial complexity of the guessing problem appears likely to be massive, even with a small number of minimal pads surrounding only the initial and final round pairs of our DES implementation.

Our approach is obviously far more obscure than previous approaches to key-hiding. At present, the stron-

gest method we can propose for estimating the security of our approach is to provide a web site at which sample DES encryption and decryption functions are provided for known keys, plus challenge implementations with unknown keys (the objective being to identify the unknown keys). As of the date of this conference, this site (***identifying URL suppressed***) is available. We are confident that our implementation techniques will stand up well to concerted attacks: experience should indicate decisively whether our confidence is well-founded.

Conclusion

The state of the art in software tamper-proofing and obfuscation in general, and hiding of cryptographic keys in software in particular, has been quite weak. We have described an approach which is far more secure at the cost of increased overhead, and suggested applications for it. We note that metrics for measuring the security of such technologies represent a vast unexplored area in computational complexity. Accordingly, we invite challengers to attempt to crack our technology for hiding cryptographic keys in software. Facilities for such attempts will be provided at our web site (`http://www.cloakware.com`).

References

[AHI 1987] Niv Ahituv, Yeheskel Lapid, and Seev Neumann. 1987. *Processing encrypted data.* Communications of the ACM 30(9), Sept. 1987, pp. 777-780.

[AUC 1996] David Aucsmith and Gary Graunke. 1996. *Tamper-resistant software: an implementation.* Proceedings of the First International Workshop on Information Hiding, Cambridge, UK.

[COL 1998-1] Christian Collberg, Clark Thomborson, and Douglas Low. 1998. *Manufacturing cheap, resilient, and stealthy opaque constructs.* ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January, 1998.

[COL 1998-2] Christian Collberg, Clark Thomborson, and Douglas Low. 1998. *Breaking abstractions and unstructuring data structures.* IEEE International Conference on Computer Languages, 1998.

[FIPS 46-3] FIPS publication 46-3. PDF available at `http://csrc.nist.gov/fips/`. *Note:* Description of DES begins on page 8.

[MARX 1998] Philipp Wilhelm Marx. 1998. *Module for the protection of software.* U.S. Patent 5,805,802.

[OST 1992] Rafail Ostrovsky and Oded Goldreich. 1992. *Comprehensive software protection system.* U.S. Patent 5,123,045.

[SCH 1996] Bruce Schneier. 1996. *Applied Cryptography.* ISBN 0-471-11709-9. John Wiley & Sons. *Note:* An extensive discussion of DES is found on pp. 265-294.